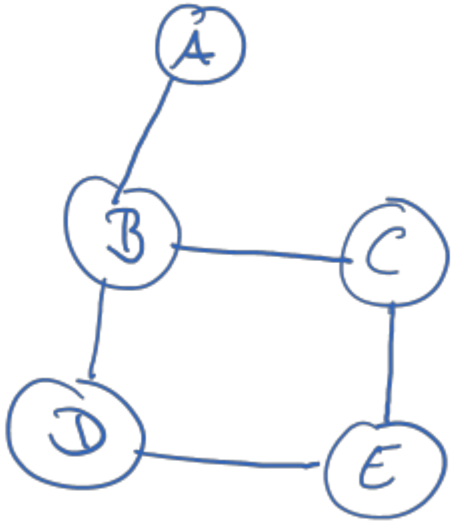
# Graphs

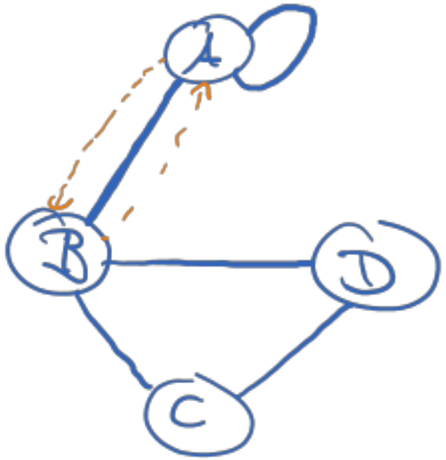
# What is a Graph?

- Graphs consist of
  - Set of nodes (or vertices)
  - Set of links (or edges)
- Two nodes connected with direct edge are
  - Neighbors or adjacent nodes

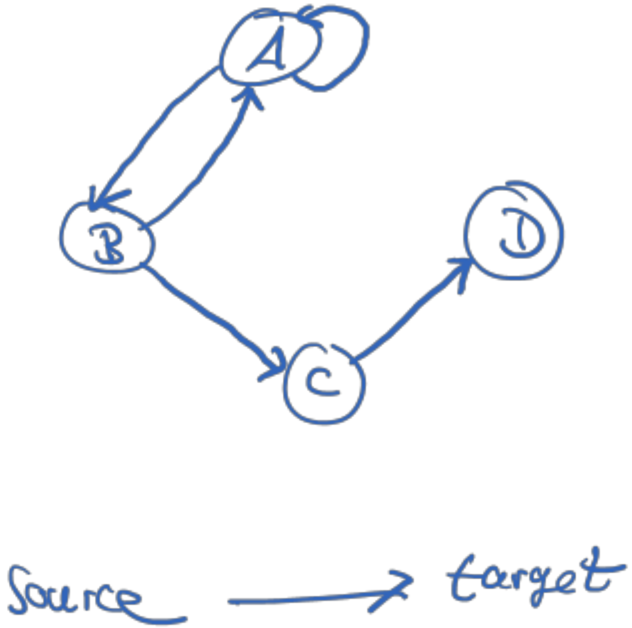


# Two Types of Graphs

undirected

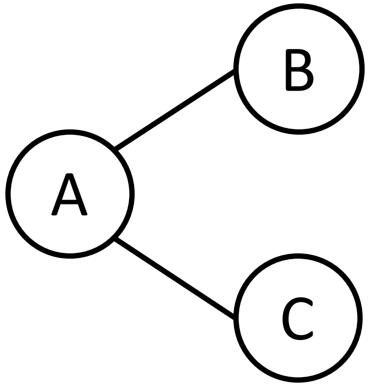



directed



# Degree of a Node

The number of edges connected to a node.




$deg(A) = 2$

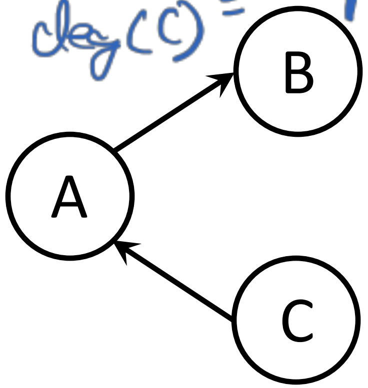
$deg(B) = 1$

$deg(C) = 1$

$deg(A) = 2$

$deg(B) = 1$

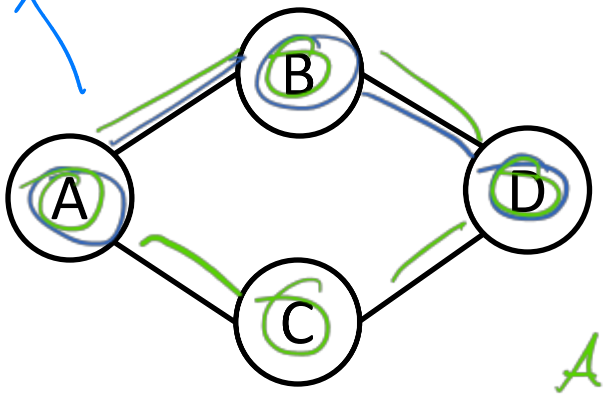
$deg(C) = 1$




| | |
|---|---|
| $indeg(A) = 1$ | $outdeg(A) = 1$ |
| $indeg(B) = 1$ | $outdeg(B) = 0$ |
| $indeg(C) = 0$ | $outdeg(C) = 1$ |

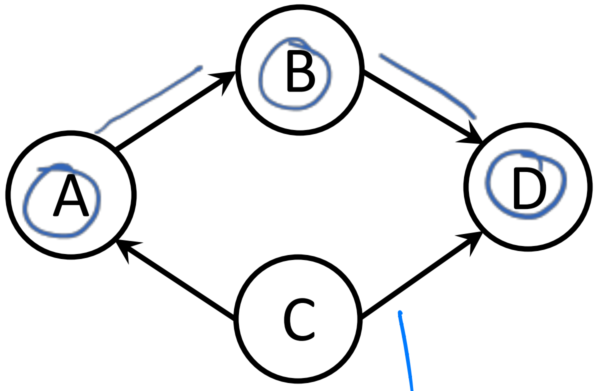
# Paths in a Graph

Cyclic graph




ABD

ABDCA




ABD

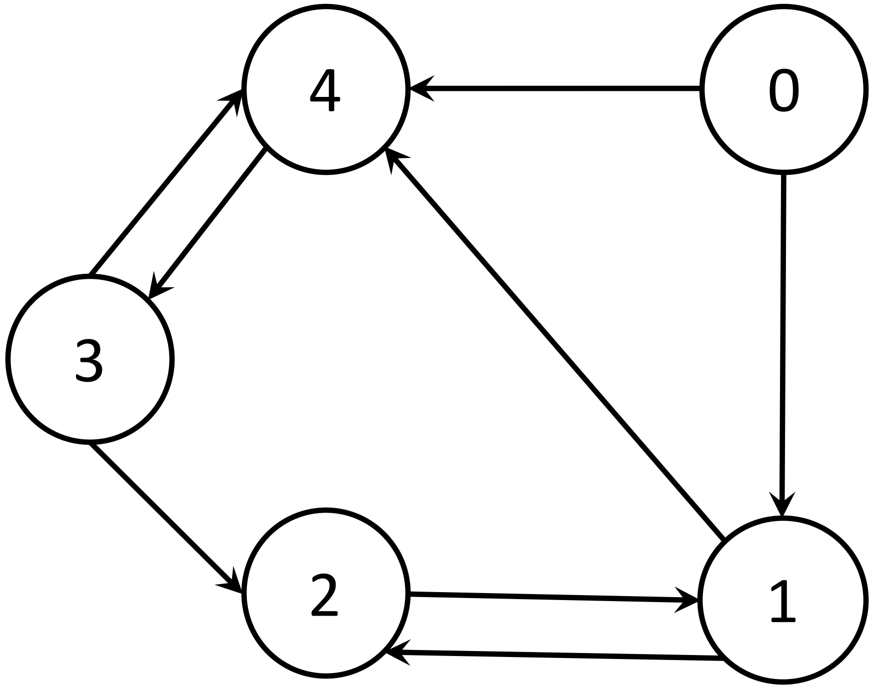
Acyclic graph

- Cycle: A path that returns to a previously visited node
- Cyclic graph: contains at least one cycle
- Acyclic graph: contains no cycle

( ☆ There are 2 ways to represent the graph )

# ① Adjacency Matrix Example 1

Usually represented as 2-D array in Java

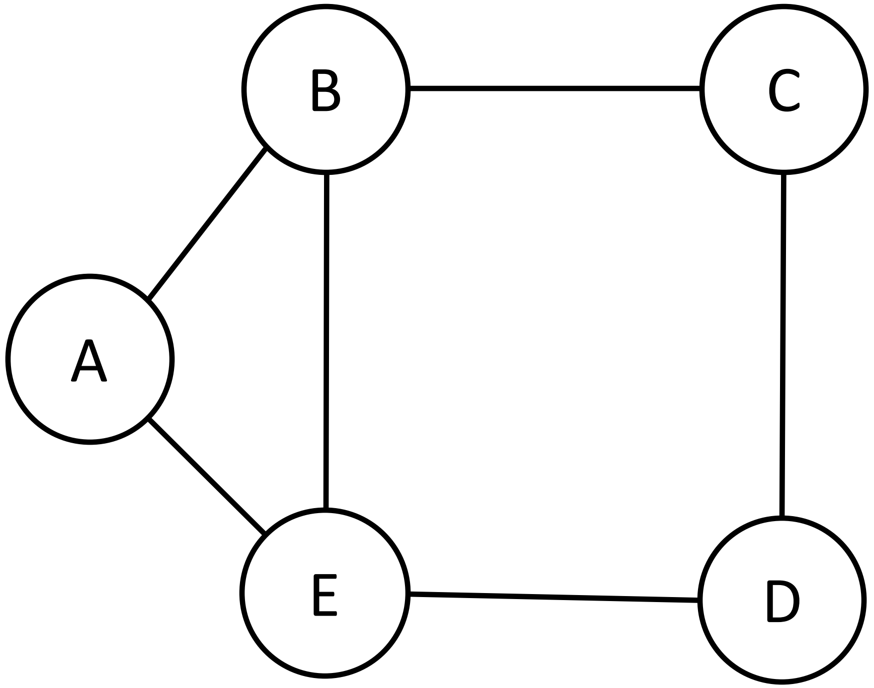



columns: target

rows: source

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | F | T | F | F | T |
| 1 | F | F | T | F | T |
| 2 | F | T | F | F | F |
| 3 | F | F | T | F | T |
| 4 | F | F | F | T | F |

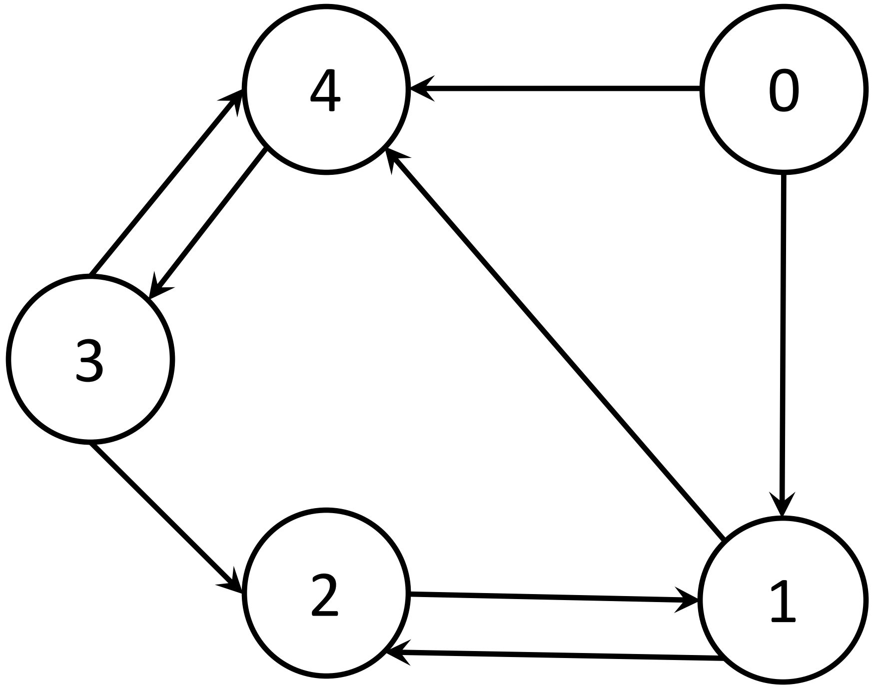
# Adjacency Matrix Example 2




| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | F | T | F | F | T |
| B | T | F | T | F | T |
| C | F | T | F | T | F |
| D | F | F | T | F | T |
| E | T | T | F | T | F |

undirected: symmetric at diagonal

# 2 Adjacency List Example 1

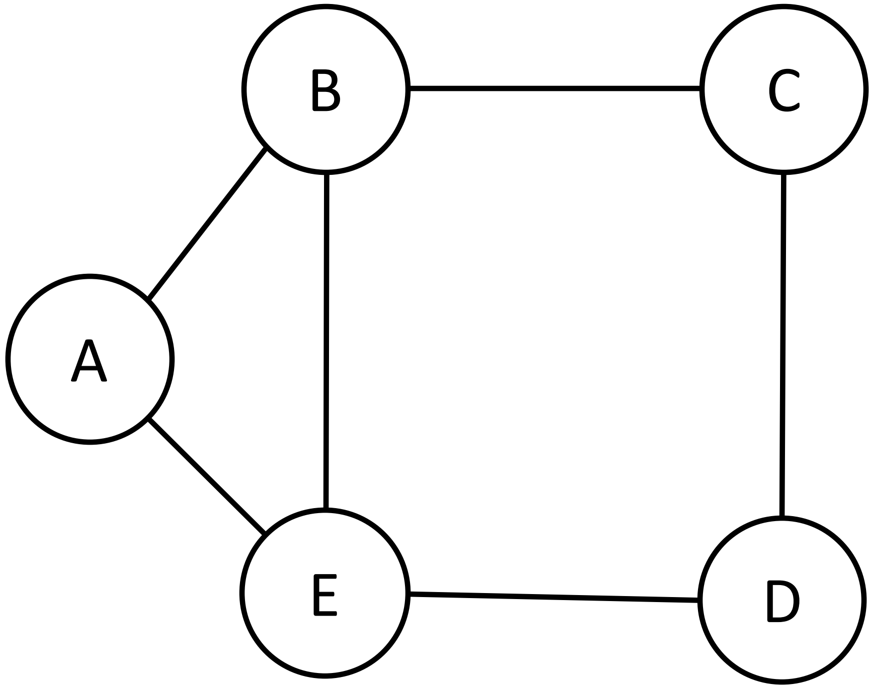



(★ List of neighbors of the selected node)

- **0:** 1, 4
- **1:** 2, 4
- **2:** 1
- **3:** 2, 4
- **4:** 3

(Here, it will have 5 lists for each of the node)

# Adjacency List Example 2




A: B, E

B: A, C, E

C: B, D

D: C, E

E: A, B, D

# Implementation of Graphs

node type

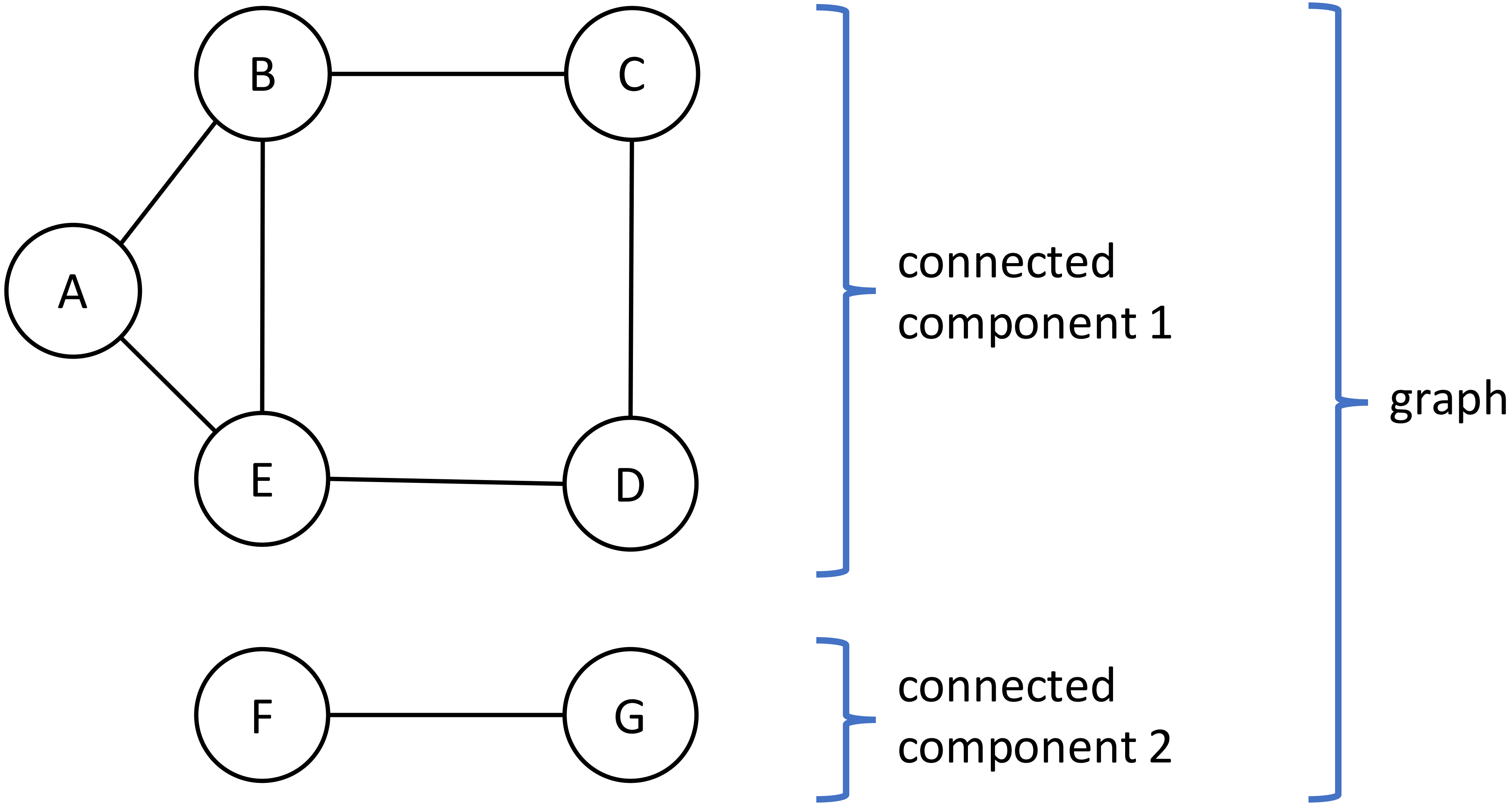
```
public class Graph <T> {
        boolean[][] adjacencyMatrix;
        Map<T,Integer> nodeIndices;

        OR

        Map<T,Graphnode<T>> vertexTable;
        protected class Graphnode<T> {
                protected T data;
                protected List<Graphnode<T>> adjacencyList;
        }
}
```

# Graph Traversals

# Graph Traversals

- Goal: Traverse graph by crossing edges to visit each node exactly once
- Points to consider
  - need to pick starting node
  - nodes might be unreachable from starting node
  - cycles can lead to infinite loops

# Unreachable Nodes: Connected Components



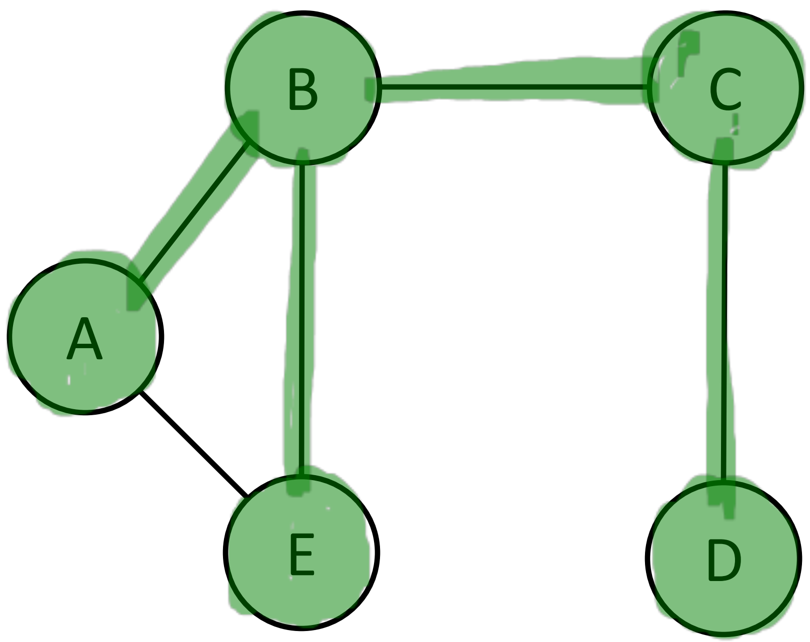
# Detecting Cycles

- Detect cycles to avoid visiting nodes multiple times
- Strategy: check if node is unvisited before visiting it
  - need to keep track of visited nodes:
  - either with Boolean field in node type, or
  - additional data structure to keep track of visited nodes

# Depth First Traversal

| adj. matrix | adj. list |
| --- | --- |
| for each call to DFT: V for loop iters. $\Rightarrow V*V$ | across V calls to DFT: E for-loop iters: $\Rightarrow V+E$ |

Assume: all vertices are marked unvisited at start




starting node

V calls

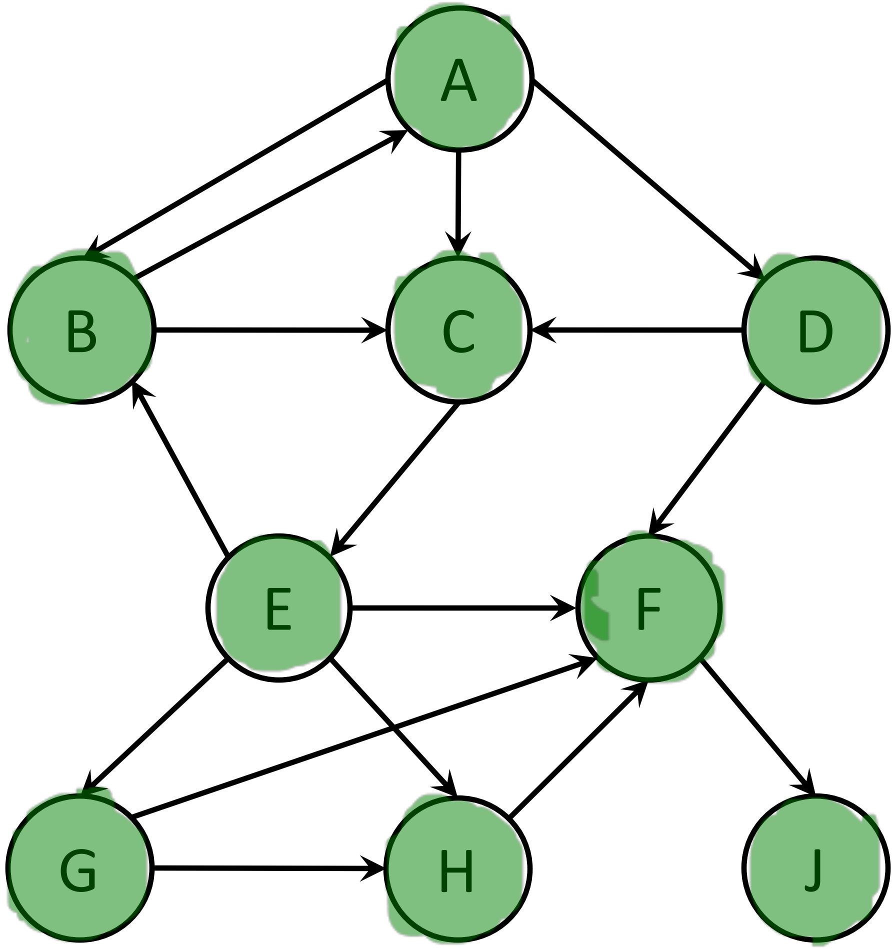
DFT(v):

mark v as visited

for each unvisited neighbor u of v:

DFT(u)

# Depth First Example




starting node: A

~~DFT(J)~~ ~~DFT(H)~~

~~DFT(F)~~ ~~DFT(G)~~

~~DFT(E)~~

~~DFT(C)~~

~~DFT(B)~~ ~~DFT(D)~~

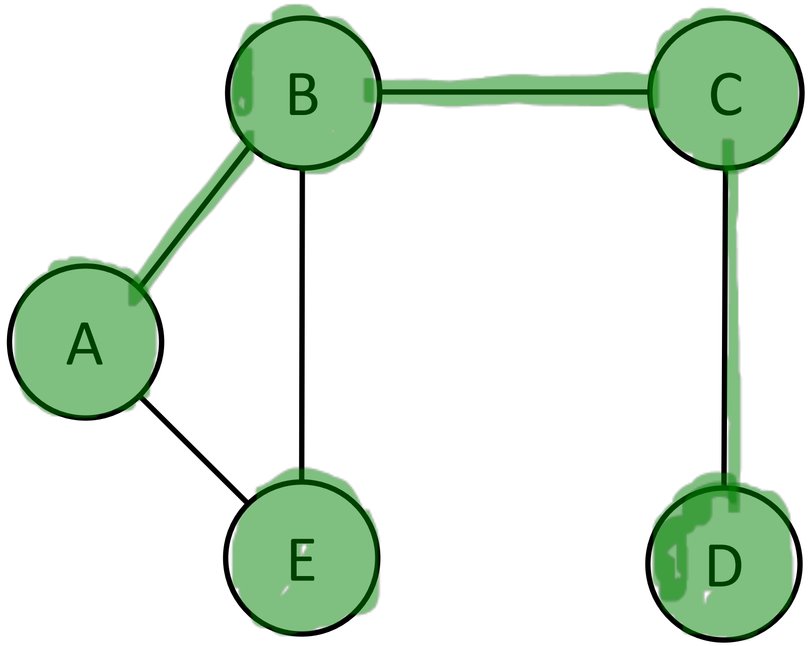
~~DFT(A)~~

visit sequence: A, B, C, E, F, J, G, H, D

# Breadth First Traversal

| adj. matrix | adj. list |
|---|---|
| for each while iter: V for-loop iters $\Rightarrow V*V$ | across V while iters: E for-loop iters $\Rightarrow V+E$ |

Assume: all vertices are marked unvisited at start

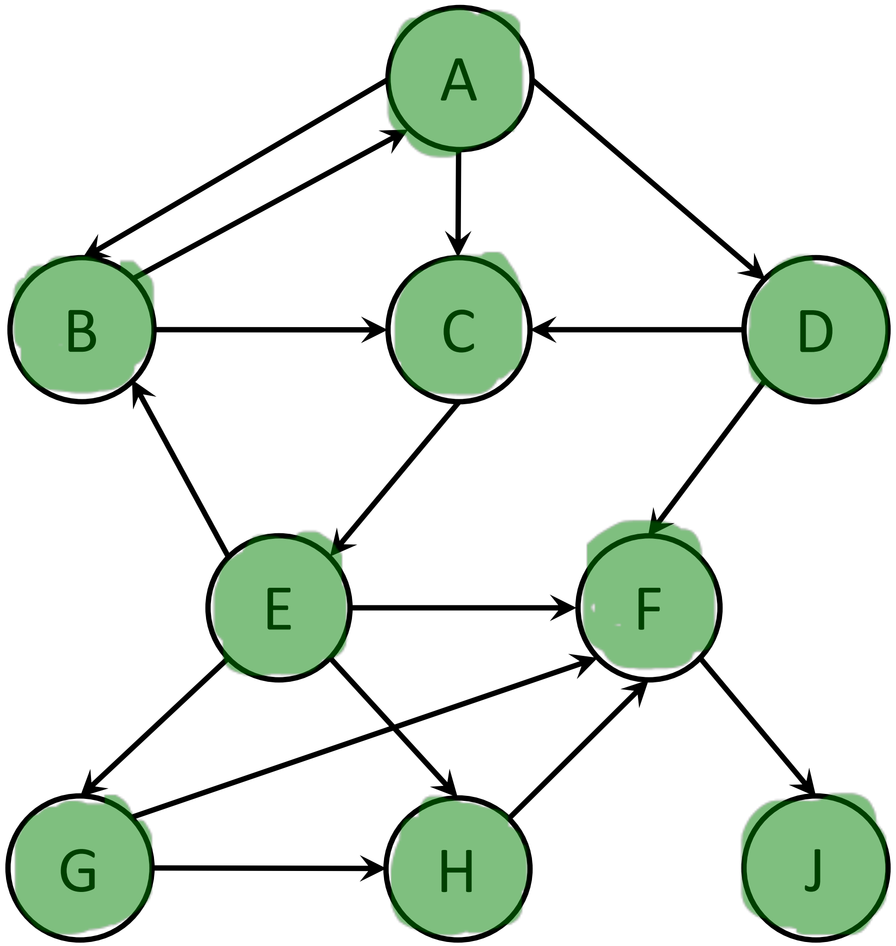



starting node

```
BFT(v):
  q = new Queue()
  mark v as visited
  q.enqueue(v)
  while (!q.isEmpty()):
     c = q.dequeue()
     for each unvisited neighbor u of c:
        mark u as visited
        q.enqueue(u)
```

V iterations

# Breadth First Example




starting node: A

queue:

<- out <- in

~~A~~, ~~B~~, ~~C~~, ~~D~~, ~~E~~, ~~F~~, ~~G~~, ~~H~~, ~~J~~

visit sequence: A, B, C, D, E, F, G, H, J

# Time Complexity

V : # of nodes in graph

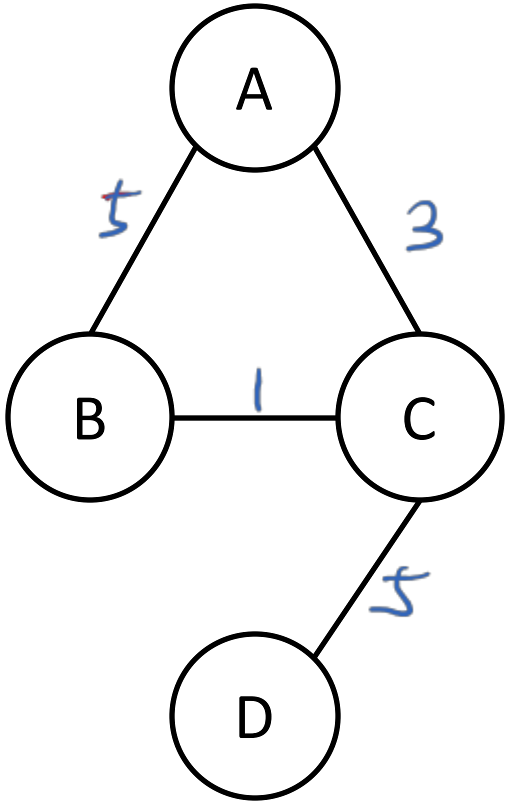
E : # of edges in graph

| Depth First | | Breadth First | |
|---|---|---|---|
| adjacency matrix | adjacency list | adjacency matrix | adjacency list |
| $O(V^2)$ | $O(V+E)$ | $O(V^2)$ | $O(V+E)$ |

# More Graph Terminology

# Weighted Graphs

Weighted edges assign a cost or weight to each edge.




```
double[][] adjacencyMatrix;
```

```
List<Graphnode<T>> adjacencyList;
List<Double> edgeWeights;
```

# Cost of a Path

- For weighted graphs: sum of edge weights on path

$$A \xrightarrow{3} B \xrightarrow{4} C$$

$$\text{cost}(ABC) = 3+4=7$$

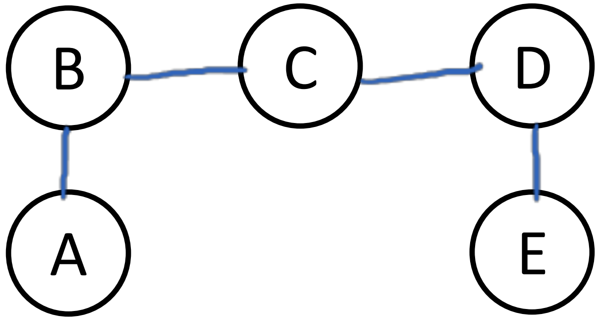
- For unweighted graphs: length of path (# of edges)

$$\text{cost}(CDE) = 2$$

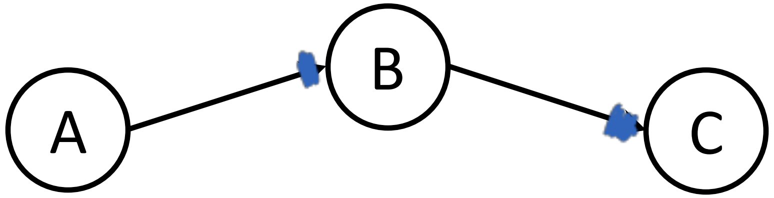
# Connected Graph

(It means you can arrive at any node from any other node)

- In a connected graph, a path exists between every pair of nodes.



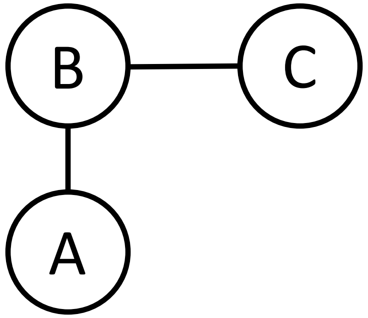

- For directed graphs:
  - strongly connected: a path exists between every pair of nodes with edge directions respected
  - weakly connected: a path exists between every pair of nodes with edge directions ignored



# Subgraphs

- G’ is a subgraph of G if:
  - The set of nodes of G’ is a subset of the nodes of G, and
  - The set of edges of G’ is a subset of the edges of G.

$G_1$




$G_2$